SpeechBuilder: Facilitating Spoken Dialogue System Development

by

Eugene Weinstein

S.B., Massachusetts Institute of Technology (2000)

Submitted to the Department of
Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in
Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 23, 2001

Author ....
Department of
Electrical Engineering and Computer Science
May 23, 2001

Certified by ....
James R. Glass
Principal Research Scientist
Thesis Supervisor

Accepted by ....
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

SpeechBuilder: Facilitating Spoken Dialogue System Development

by

Eugene Weinstein

Submitted to the Department of
Electrical Engineering and Computer Science
on May 23, 2001, in partial fulfillment of the
requirements for the degree of
Master of Engineering in
Electrical Engineering and Computer Science

Abstract

SpeechBuilder is a suite of tools that helps facilitate the creation of mixed-initiative spoken dialogue systems for both novice and experienced developers of human language applications. SpeechBuilder employs intuitive methods of specification to allow developers to create human language interfaces to structured information stored in a relational database, or to control- and transaction-based applications. The goal of this project has been both to robustly accommodate the various scenarios where spoken dialogue systems may be needed, and to provide a stable and reliable infrastructure for design and deployment of applications. SpeechBuilder has been used in various spoken language domains, including a directory of the people working at the MIT Laboratory for Computer Science, an application to control the various physical items in a typical office environment, and a system for real-time weather information access.

Thesis Supervisor: James R. Glass
Title: Principal Research Scientist

Acknowledgments

First, I would like to sincerely thank my thesis advisor, Jim Glass, for inspiring this project and working through it with me all the way to the completion to this thesis. Without his guidance, perseverance, and patience, I would have never been able to finish this work.

In addition, I would like to thank all of the staff, visiting scientists, and students that have helped out in this work. Just about every person in the Spoken Language Systems Group has contributed to creating SpeechBuilder, many of them making quite a significant impact on the system. Of these people, I would like to specifically thank Issam Bazzi, Scott Cyphers, Ed Filisko, TJ Hazen, Lee Hetherington, Mikio Nakano, Joe Polifroni, Stephanie Seneff, Lynn Shen, Chao Wang, and Jon Yi. I would also like to thank Jef Pearlman, who built SpeechBuilder's predecessor system, SLS-Lite, as part of his Master's thesis research.

Finally, I would like to thank my family and friends. My father Alexander, mother Alla, and sister Ellen have all given me the encouragement and care without which I could not have made it through MIT. I am also very grateful to my friends for their support and encouragement, especially in the home stretch of this thesis. Specifically, I would like to thank my friend Jean Hsu for proofreading this paper.

This research was supported by DARPA under contract N66001-99-1-8904 monitored through Naval Command, Control and Ocean Surveillance Center and under an industrial consortium supporting the MIT Oxygen Alliance.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The process of designing, implementing, and deploying a mixed-initiative spoken dialogue system is non-trivial. Creating a competent and robust system that affords its users significant freedom in what they can say and when they can say it has always been a difficult task. Building these systems is time-consuming and tedious for those who can be considered experts in this field, and essentially impossible for those without any experience. SpeechBuilder, which has been developed as part of this thesis research, is a suite of tools that simplifies and streamlines this process. SpeechBuilder aims to provide methods of configuring application specifics that even a complete novice developer can tackle, while maintaining the power and flexibility that experienced developers desire.

1.1 Motivation

Among the many factors motivating the creation of SpeechBuilder, the most prominent were 1) the benefits of making spoken dialogue system development accessible to novices, 2) the opportunities for data collection, and 3) the need to provide an efficient way for experts to rapidly build spoken dialogue systems.

Over the past decade, the Spoken Language Systems Group at the MIT Laboratory for Computer Science has been actively developing the human language technologies necessary for creating conversational human-machine interfaces. In recent years

several systems have been publicly deployed on toll-free telephone numbers in North America, including systems providing access to information about weather forecasts, flight status, and flight schedules and prices [37, 28]. Although these applications have been successful, there are limited resources at MIT to develop a large number of new domains. Since it is the development of new domains that often exposes weaknesses or deficiencies in current technology, the scarcity of resources is a stumbling block for the advancement of the state of the art. SpeechBuilder attempts to address this problem by streamlining the application development and deployment process, while maintaining use of the same technologies that are used to build dialogue systems by hand.

Even with tools such as SpeechBuilder, the variety of spoken dialogue systems that experts can propose and create is limited. However, the number of people that may be interested in creating applications using spoken language technology is virtually unlimited, as are their ideas. Thus, it is naturally desirable for users of all skill levels and backgrounds to be able to experiment with this technology. SpeechBuilder addresses this problem by utilizing intuitive methods of specification and easily-managed deployment schemes to allow novice users to build spoken dialogue systems.

Lastly, enabling a wide range of developers to implement and deploy various novel domains presents significant opportunities for data collection. Applications built using SpeechBuilder will provide raw data that will help improve the acoustic-phonetic models used in recognition at MIT. This is especially useful because it is possible that SpeechBuilder domains will present data from a wider variety of speakers and acoustic environments than that of the systems deployed to date.

1.2 Goals

In this work the goals have been 1) to robustly accommodate the various scenarios where spoken dialogue systems may be needed, and 2) to provide a stable and reliable infrastructure for design and deployment of applications. Success in this project would

be to achieve these goals while maintaining ease of use for developers seeking to build domains.

In past work dealing with facilitating spoken language system development [22], the predominant model for SpeechBuilder-type tools has been to provide a speech interface to a pre-existing application, the functionality of which appears as a black box to the human language technologies. However, over the course of developing SpeechBuilder it became clear that many domains that were of interest were information access applications (e.g. weather, flight information, stock quotes, directions). Thus, it became desirable to develop a SpeechBuilder application model that involves configuring a speech-driven front end to a set of structured information. SpeechBuilder addresses both problems in what have become two separate application development approaches and methodologies.

SpeechBuilder attempts to enable developers to build conversational systems that are comparable in quality and robustness to hand-built systems. This effort involves many factors, such as providing effective mixed-initiative dialogue strategies, robust language models for recognition and natural language parsing grammars, and the tools for modeling the often complex relations between information concepts.

1.3 Approach

The approach that has been used in SpeechBuilder is to leverage the basic technologies that have been used to hand-build various demonstration dialogue systems within the galaxy framework for spoken language system development [27]. SpeechBuilder uses information specified by the developer to configure human language technology (HLT) components based on the specifications and constraints provided by the developer. Since SpeechBuilder aims to allow developers to create spoken dialogue systems comparable to those that have been hand-built in the past, it is natural that SpeechBuilder should utilize much of the same human language technology framework as the hand-built domains. In addition to this intuitive connection between SpeechBuilder domains and these technology components, there

are several other reasons why this approach has been selected.

First, significant effort has been devoted in the past at MIT to improving technology in dialogue system architecture [27], speech recognition [11], language understanding [24], language generation [1], discourse and dialogue [28], and, most recently speech synthesis [36]. Employing these HLT components minimizes duplication of effort, and maximizes SpeechBuilder's flexibility to adopt technical advances made in these areas, which may be achieved in efforts entirely disjoint from SpeechBuilder development.

Second, since SpeechBuilder uses the full and unconstrained set of technology components, domains created using SpeechBuilder can eventually scale up to the same level of sophistication as that of the domains that have been hand-built at MIT. In fact, this enables expert developers to use SpeechBuilder as a "springboard" to bypass the often tedious and stressful effort of going through the initial stages of spoken language application development.

Finally, since SpeechBuilder configures the core HLT components to accommodate user constraints and specifications, it is quite likely that the unforeseen scenarios created by developers building various domains will expose deficiencies in the HLT's and thus drive potential improvements. In addition, since SpeechBuilder aims to build portable domains with independently functioning components, it can help stimulate improvements to the abstraction and portability of these components.

1.4 Terminology

As mentioned in Section 1.2, SpeechBuilder is modeled to allow for creation of applications in the various scenarios for which speech-driven applications may be needed. Currently, SpeechBuilder has two fundamentally different models for domain structure to accommodate these scenarios.

The first model assumes that the developer desires to "speech-enable" an application; that is, there is an application in existence, and the developer would like to create a spoken language interface to that application. This is referred to as the

control model, because it can be used to create applications controlling various real-world devices or software extensions. An example of an application that can be implemented via the control model is the office device control domain, which allows a person to use speech to control the various physical devices in their office. Another example is a domain allowing the booking of flight reservations using spoken dialogue – this falls under the control model because of the complex transaction control involved in completing a request.

The second model is designed for developers who want to provide a spoken interface to structured information stored in a database. This is referred to as the **info model**. Examples of domains that fall under this model include stock quotes, weather information, schedule access, and radio station information.

1.5 Outline

Chapter 2 discusses previous work on facilitating the design and development of spoken dialogue systems. Previously implemented technologies are compared and contrasted to SpeechBuilder.

Chapter 3 gives an overview of SpeechBuilder's architecture. It discusses how the various human language technology components are used within SpeechBuilder in both the control and the info models. This chapter also describes the web interface that developers use to configure domain specifics.

Chapter 4 describes the knowledge representation used by SpeechBuilder. It addresses the key-action linguistic concept representation inherited from the SLS-Lite system, the data concept representation used in the info model, and the response specification facility used in SpeechBuilder.

Chapter 5 discusses some of the more non-trivial issues that have come up during SpeechBuilder development and implementation. Specifically, it discusses issues in speech recognition and natural language understanding, example generation, text-to-phone conversion, and overall infrastructure.

Chapter 6 describes the current operating status of SpeechBuilder and gives

an overview of some of the domains that have been built by various developers inside and outside of MIT.

Chapter 7 summarizes the thesis and proposes some future work that would improve SpeechBuilder's infrastructure and robustness.

Chapter 2

Background

There are a number of efforts, present both in industry and in academia, to provide tools for development of systems using spoken language and the formalisms for defining such systems. SpeechBuilder and its predecessor system, SLS-Lite, appear to be among a small set of toolkits for development of mixed-initiative natural language environments. This chapter presents the efforts to date and compares them to the work done in this thesis.

2.1 SLS-Lite

SLS-Lite is the prototype system that preceded SpeechBuilder [22]. SpeechBuilder inherits much of its philosophy, as well as many of its features (such as the linguistic concept representation), from SLS-Lite.

The SLS-Lite system focused on building spoken language interfaces to applications that might normally be driven with other modalities. SLS-Lite was the first implementation of the framework linking recognition and understanding components to a back-end application that implements domain-specific functionality and turn management. This framework is still present as the "control model" in SpeechBuilder.

SLS-Lite introduced the methodology of allowing developers to configure domain specifics using intuitive methods of specification. Developers specified linguistic

constraints and natural language templates by giving example user queries (actions) and identifying concepts (keys). This is described in more detail in Chapter 4.

Finally, much of the code base for the SLS-Lite system is present in SpeechBuilder. SpeechBuilder introduces a more sound deployment infrastructure, and a new mode of operation (info model) utilizing all of the major galaxy components. However, SLS-Lite is the foundation framework on which SpeechBuilder was created, therefore much credit for the work leading up to this thesis research should be given to Jef Pearlman and Jim Glass, who designed and implemented the original SLS-Lite system [22].

2.2 VoiceXML

VoiceXML, the Voice eXtensible Markup Language, is a language for describing human-machine interactions utilizing speech recognition, natural language understanding, and speech synthesis. VoiceXML was developed by a forum consisting of various companies interested in spoken dialogue systems (such as AT&T, IBM, Lucent and Motorola) and was later standardized by the World Wide Web Consortium [33]. VoiceXML seeks to provide a medium for streamlined speech-driven application development. However, the philosophy of VoiceXML differs from that of SpeechBuilder in several key areas.

Primarily, SpeechBuilder is orthogonal in purpose to VoiceXML. VoiceXML is a mark-up language which allows developers to specify the dialogue flow and the language processing specifics of a speech-driven application. SpeechBuilder is more of an abstraction layer between the developers of spoken language domains and the technology that implements these domains. In fact, it is entirely possible that in the future, SpeechBuilder will able to generate VoiceXML representations of the domains built by developers. However, at the current time, SpeechBuilder development has focused around deploying domains in the galaxy framework [27] (galaxy has been adopted by the DARPA Communicator Program as the reference architecture for design of mixed-initiative spoken language domains [26]). In contrast,

the VoiceXML standard is focused on more directed-dialogue applications, using very constrained language parsing grammars. Since more flexible mixed-initiative systems are harder to create than directed-dialogue domains, there is more of a need for tools such as SpeechBuilder to help developers accomplish this task.

While SpeechBuilder is orthogonal to VoiceXML in purpose, it is important to compare the technologies currently used by SpeechBuilder to those present in systems specified with VoiceXML. VoiceXML uses Java Speech Grammar Format (JSGF) [14] context-free grammars to define natural language understanding templates, with mechanisms for parsing poorly formulated queries. SpeechBuilder domain recognizers are currently configured to use a hierarchical n-gram language modeling mechanism (a more robust model for multi-concept queries) with backoff to robust parsing (to help identify concepts when the user's utterance contains useful information but does not match the parsing grammars exactly). In addition, as mentioned above, VoiceXML is primarily a protocol for describing human-machine interactions in a predefined finite-state sequence of events (this is known as "directed dialogue"). In contrast, SpeechBuilder allows developers to build mixed-initiative applications, where the user can say any in-domain utterance at any time in the conversation.

Finally, since VoiceXML is a specification language and not a complete system for application configuration and deployment such as SpeechBuilder, an implementation utilizing VoiceXML requires the configuration of third-party speech recognition and synthesis components. Several groups have implemented systems based on the VoiceXML framework. It is appropriate to compare the functionality and flexibility of these systems to that of SpeechBuilder, and this is done in the next two sections of this chapter.

2.3 Tellme Studio

Tellme Studio is one implementation of the VoiceXML framework. It is similar to SpeechBuilder in that it uses a web interface to configure application specifics.

However, it differs fundamentally from SpeechBuilder in that it requires developers to write JSGF grammars and call flow automata as part of the VoiceXML document defining their domains. Thus, using Tellme Studio requires a certain amount of expertise, and can involve a significant learning curve for those developers not familiar with writing language parsing grammars. SpeechBuilder's web interface, in contrast, uses example-based specification methods to define grammars and other domain specifics. In fact, any developer can configure all of the aspects of the application using the web interface; while more experienced developers are able to modify domain specifics by manipulating the raw XML representation of their domains.

SpeechBuilder is similar to Tellme Studio in that it allows developers to deploy applications using the the respective development site's HLT framework and telephony infrastructure. In addition, both systems allow for local installations for advanced developers who would like to migrate mature applications to a deployment infrastructure at the developer's actual physical location.

2.4 SpeechWorks and Nuance

SpeechWorks [32] and Nuance [21], two of the market leaders in spoken dialogue system development, have both begun efforts to simplify development of voice-driven systems for their customers. SpeechWorks has created the SpeechSite product, which is a pre-packaged solution for company information retrieval and presentation using a speech-driven interface. SpeechSite is available commercially at a price level primarily accessible to medium- to large-size companies. SpeechSite is similar in purpose to some applications that may be built using SpeechBuilder, but is designed for a specific domain and therefore can not be directly compared to SpeechBuilder. In addition, SpeechWorks has implemented a "VoiceXML browser," which is a software package which deploys realizations of VoiceXML domains within the SpeechWorks speech recognition and synthesis framework.

Nuance has developed V-Builder, which is a commercial product designed for implementing VoiceXML-specified speech-driven domains. V-Builder provides a graphi-

cal user interface to allow developers to build applications compliant with the VoiceXML protocol. However, domains created using V-Builder are prone to the same directed dialogue call flow and grammar limitations as Tellme Studio (see section 2.3).

2.5 Others

Philips has created a unified platform for speech application development called SpeechMania [23]. SpeechMania allows developers to define spoken language applications using a language called High Level Dialogue Description Language (HDDL). HDDL gives the developers total control over the flow of the dialogue, and provides connections to language technology components such as speech recognition, natural language parsing, and synthesis. In this, SpeechMania is very similar to SpeechBuilder (in the control model). In contrast to SpeechBuilder, SpeechMania does not utilize intuitive methods of linguistic specification, and therefore is not as accessible to novice developers and those looking to quickly create a working application. However, it is probably a more robust solution than SpeechBuilder for experienced developers of dialogue systems who want to have total control over the dialogue and HLT components. Because SpeechMania allows the developer to control the call flow using HDDL, it is not restricted to directed dialogue as VoiceXML systems are.

The Center for Spoken Language Understanding (CSLU) at the Oregon Graduate Institute of Science and Technology has created a toolkit for the design of spoken dialogue systems [6]. The CSLU Toolkit utilizes example-based methods of specifying natural language understanding and generation templates, and uses a graphical interface to configure the dialogue manager. These features make the CSLU Toolkit accessible to developers of all levels of expertise; however, it is still constrained to building only directed-dialogue systems.

Chapter 3

Architecture

SpeechBuilder makes use of the galaxy framework [27], which is used by all of the spoken dialogue systems developed at MIT (see Section 1.3). galaxy is a highly flexible and scalable architecture for dialogue systems, and it serves as the DARPA reference architecture for building research systems utilizing human language technologies [26]. The first section of this chapter describes the HLT components that are used within SpeechBuilder and how the galaxy framework is used to unify them. The second section describes the web-based interface that the developer uses to configure domains.

3.1 Technology Components

The technology components for SpeechBuilder domains created within the control model are configured quite differently from those using the info model (Section 1.4 defines these terms). Specifically, control model domains use a very limited language generation server and have no built-in components to manage dialogue, discourse, or information retrieval. Info model domains utilize a more complete set of HLT components, similar to dialogue systems previously built within the galaxy framework [37, 28]. However, several components are present in both kinds of domains, and are configured largely as they are in SLS-Lite [22], the predecessor system to SpeechBuilder. This section presents the major technology components organized

by which domain model(s) use them, and ordered sequentially as they are used during a turn of an application. Figures 3-1 and 3-2 illustrate the HLT architecture of a SpeechBuilder domain, in the info model and the control model, respectively.

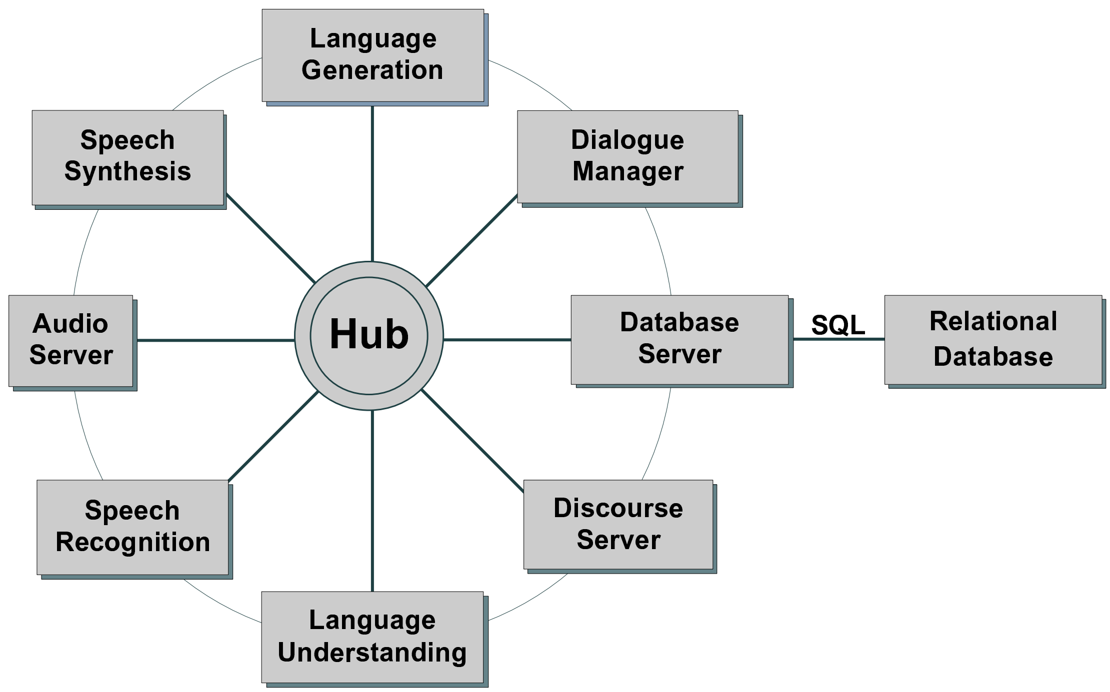

Figure 3-1: Info model SpeechBuilder configuration utilizing full set of GALAXY components

3.1.1 Components Used Both in Control and Info Models

Hub

The GALAXY hub manages the communication between the HLT components in the application. While some GALAXY components may be designed to operate directly on other components' output (e.g. the natural language understanding component takes in N-best hypotheses from the recognizer), each component is written so that it operates completely independently of the rest of the system. Thus, a central control module is necessary to control the information flow between the various HLT components. The hub serves as this module.

The hub is a programmable server which uses a rule-based language to specify

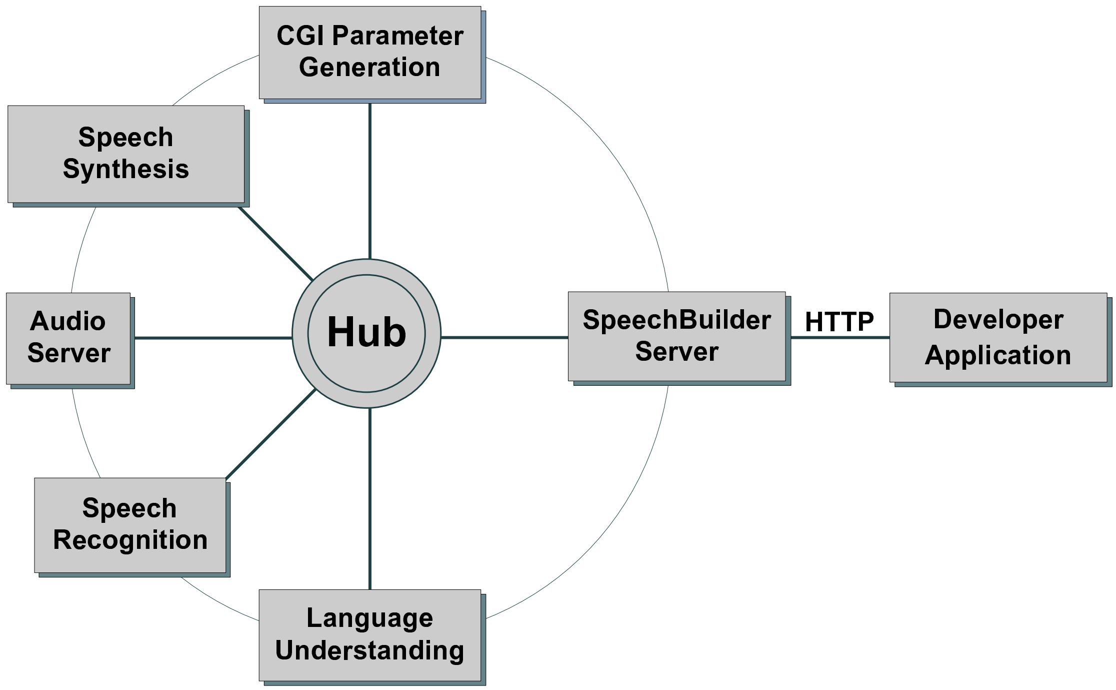

Figure 3-2: Control model SpeechBuilder configuration utilizing limited set of galaxy components

hub behavior in response to various events. The SpeechBuilder hub runs a special program similar to that of the main systems built within the galaxy framework.

Audio

SpeechBuilder applications use an audio server to control the hardware that accepts and digitizes the acoustic signal of the utterance. The most common mode of operation for a SpeechBuilder domain is to accept utterances over a telephone line. This is handled by the audio server in *telephony* mode.

In addition to being able to process telephone audio input, the audio server can function in *local-audio* mode, which allows it to process utterances coming in over a microphone connected to a computer's sound card. This allows developers to deploy installations of SpeechBuilder domains local to their own hardware, given that they have access to galaxy software.

Speech Recognition

SpeechBuilder domains use the summit speech recognizer [11] to transform the speech signal into a list of N-best hypothesis strings for the spoken utterance. The recognizer uses a hierarchical n-gram [22, 18, 34] as a statistical model for the kind of sentences that it expects from the user. The linguistic information given by the developer is used to generate the recognition grammars and to train the statistical recognition model (n-gram). Generic telephone acoustic models are used for processing the speech signal. These models are based on over 100 hours of training data collected primarily from the jupiter [37], voyager [13, 27], pegasus [30], and mercury [28] domains.

The SpeechBuilder recognizer uses a new out-of-vocabulary (OOV) word model [2]. Out-of-vocabulary words, when not properly modeled, can seriously impair recognition. The reason for this is that an unknown word in a sentence is not only misrecognized, but can also result in deletion and substitution errors elsewhere in the utterance. Since SpeechBuilder recognizers are usually trained from a small set of example sentences, this problem is especially pronounced in SpeechBuilder domains. The SpeechBuilder OOV models allow for a recognition hypothesis to contain the tag "`<unknown>`" where an unknown word is detected. Usually in these cases, at least partial information about the meaning of the utterance can be extracted by the natural language understanding component.

Natural Language Understanding (NLU)

The NLU component takes the N-best list from the recognizer, parses each hypothesis, and picks the one with the highest recognition score that matches the NL grammar (which is configured based on developer specifications). Then it encodes the selected hypothesis sentence in a "semantic frame" – a meaning representation of the query. The NLU component used by SpeechBuilder is the tina server that is present in all of the prototype systems that have been built at MIT [24].

TINA works by matching the terminals of a hierarchical grammar to words occurring in the utterance and attempting to build up a "parse tree" of nodes (which are usually domain concepts). The semantic frame generated by TINA is a mapping of the concepts that have been identified in the utterance to their corresponding values. This process is driven by the natural language parsing grammar. Figures 3-3 and 3-4 illustrate example semantic frames. Figure 3-4 shows a semantic frame that was made according to a grammar with a hierarchical structure (as specified by the developer). Figure 3-5 illustrates the parse tree that is created when the grammar is applied to this utterance.

```
{c request
   :property "forecast"
   :weather "rain"
   :city "Boston"
   :state "Massachusetts"
   :day "Monday"}
```

Figure 3-3: Semantic frame for query "`What is the weather in Boston Massachusetts on Monday?`"

```
{c list
   :pred {p departure_time==
            :relative "before"
            :pred {p time==
                     :hour="10"
                     :xm="AM"  } } }
```

Figure 3-4: Semantic frame for query "`Are there any flights leaving before ten a m?`"

Speech Synthesis

The speech synthesis component takes the textual response made by the language generation component, and converts it to an acoustic signal resembling human speech,

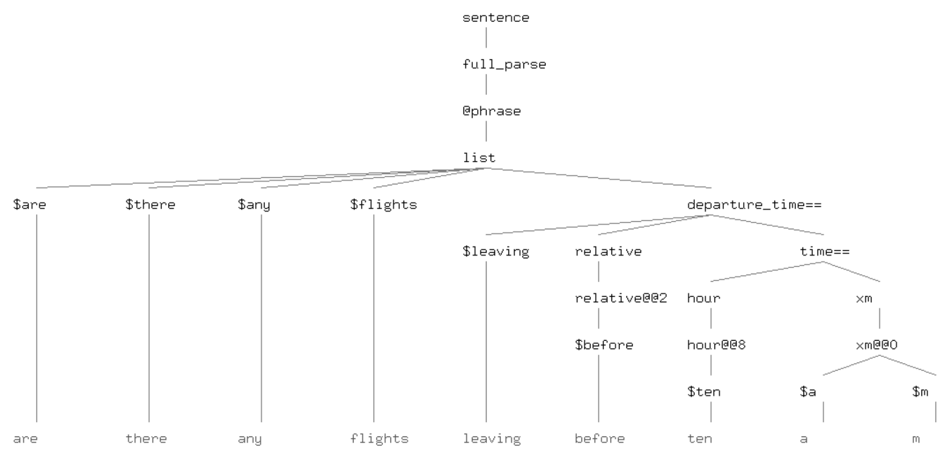

Figure 3-5: TINA parse tree for query "`Are there any flights leaving before ten a m?`"

which the audio component plays back to the user. SpeechBuilder uses DECtalk, a commercially available synthesizer [9].

Text Generation

The text, or language, generation component (implemented via the GENESIS server [1]) is responsible for creating the various text representations of the meaning encoded in the semantic frame. In the control model, text generation is used to create a CGI-encoded version of the semantic frame, which is passed to the back-end application via an HTTP GET request. Table 3.1 shows examples of utterances and their corresponding generated CGI encodings.

In the info model, the generation component takes on a much more significant role – it is actually used for generating three different outputs. The first use of generation is to create an internal "E-form" representation used by the discourse and dialogue components [28]. E-forms, like semantic frames, encode the meaning representation of the query, and can be augmented by the discourse and dialogue components based on the query's context. Figure 3-6 gives an example of an E-form. In this example, the clause (or action) name is "request_property," the output language that the next iteration of GENESIS should use is "`sql`," and the `*conditions*` item indicates that the set of rows coming back from the database should be tested for non-nullness.

Can you tell me the phone number for Bob Jones **action=request_property** **&frame=(name=Bob+Jones, property=phone)**
I will be here from nine thirty to eleven o-clock **action=ScheduleTime&frame=(time1=(hour=nine,** **minute=thirty),time2=(hour=eleven,minute=zero))**
I would like to fly from Boston to San Francisco on Wednesday morning **action=list&frame=(departure_time=(time_of_day=morning,** **weekday=wednesday),destination=SFO,origin=BOS)**

Table 3.1: Sample utterances and their CGI encodings made by the generation component.

```
{c eform
      :clause "request_property"
      :name "Victor Zue"
      :property "phone"
      :domain "SpeechBuilder"
      :*conditions* {c condition
                      :key ":num_found"
                      :value 0
                      :test "lt" }
      :out_lang "sql" } }
```

Figure 3-6: E-form for "`What is the phone number for Victor Zue?`"

The second use of the generation component is to formulate an SQL query based on a semantic frame representing a request. For example, the following SQL query would be generated based on the request "`What is the phone number for Victor Zue?`":

```
select * from speech_SLSinfo_table1 where name = 'Victor Zue'
```

The third use is to generate a response to the user which is vocalized using the speech synthesizer.

3.1.2 Components used only in Info Model

Discourse

The discourse component is responsible for filling out a query using information collected in its predecessor queries. For example, a user might say, "`What is the phone number for Joe Foo?`" and then "`What is his email address?`" To humans, it is natural to assume that the latter query is asking about Joe Foo's email address. The discourse component attempts to capture this intuition by recording and using the utterance history.

In the control model, discourse resolution is left up to the back-end application. The SpeechBuilder server can maintain basic history information, but the application is responsible for storing and using this history to evaluate queries in context.

In the info model, a discourse server component is used. This component is configured based on the concepts present in the application. It allows a query to inherit key-value pairs from predecessor queries. In addition, it allows for masking of history information when certain conditions indicate that it's illogical to inherit [10].

Dialogue Management

In the control model, there is no dialogue management within the galaxy architecture. Instead, the back-end application handles the turn management and state tracking (see section 3.1.3).

In the info model, the dialogue management server is modeled after the functionality of the main MIT systems. This server manages the interaction of the NLU component, the database server, and the language generation engine. After the query is formulated as a semantic frame and is used to make an SQL query in the database, the dialogue management component fills out the E-form using the database results.

This component is designed to handle the range of situations which can arise in database query domains. For example, when there are no matches to an SQL query in the database, this server sends a signal to the language generation component so that it can output an appropriate response to the user. Similarly, when there is more

than one match to a query, the dialogue management component fills out the E-form, so that the results can be presented in an informative, but not overwhelming, fashion to the user.

Database

The database component is a link to the data server handling SQL requests. The function of the database component is to send SQL strings to the database and to return the results as a key-value mapping similar to a flat semantic frame. The database component used in SpeechBuilder is one that has been developed for other domains. SpeechBuilder uses an Oracle database, although it would be trivial to adapt to any database accessible with SQL.

3.1.3 Component Used only in Control Model – Back-end

In the control model, the back-end application is responsible for hooking into external functionality, managing the dialogue and discourse, and generating responses. The application is connected to the galaxy components via a "back-end server." This server receives the semantic frame (encoded as CGI parameters) from the language generation component, and uses HTTP to send it to the back-end application. The back-end application processes the frame, takes any actions corresponding to the query, and returns a response to be synthesized for the user.

CGI scripts are stateless from query to query; each invocation of a script is completely independent from any other invocation. However, maintaining some historical information about previous queries is necessary for robust handling of local discourse phenomena, and for maintaining a proper dialogue. For this reason, the back-end server is designed to maintain a history string in its state variables. The back-end application script is responsible for encoding its own history and later processing it correctly.

In the info model, the database server, the dialogue manager, and the discourse components provide the functionality that the back-end server and CGI application

provide in the control model. Thus, this server is not relevant and is not used in the info model.

3.2 Web Interface

SpeechBuilder uses a web interface to allow developers to specify the details of their applications. The web interface consists of a main module containing functionality to edit the linguistic information for an application and several helper tools to edit various details.

3.2.1 Keys and Actions

The main SpeechBuilder interface module allows developers to edit the concept keys (the concepts present in the domain) and the sentence level actions (example sentences using those concepts). Figure 3-7 illustrates a developer editing the "list" action within the "mercury" domain. The developer can select existing sentences to edit or delete, or can enter new sentences. On this screen, the developer can upload an XML representation of the domain specifics, or a CSV file representing the data to be used in the domain (in the info model). The developer can also specify the URL for a CGI script with back-end functionality for the domain (in the control model).

3.2.2 Vocabulary Editing

SpeechBuilder provides developers with a tool to edit the vocabulary used for speech recognition in a domain. SpeechBuilder relies on a dictionary of about 100,000 words (LDC [16] PRONLEX) and their pronunciations to generate the vocabulary for a domain. However, often the developer introduces vocabulary words that do not appear in the main dictionary. For these words, pronunciations are generated by rule [3], and often contain errors. The vocabulary tool allows the developer to adjust the pronunciations present in the working vocabulary. This requires the developer to be familiar with the phonemic unit set used by the SUMMIT speech

recognizer.

3.2.3 Log Viewing

GALAXY maintains a comprehensive log for each domain built and deployed using SpeechBuilder. Looking at these logs allows dialogue system developers to improve their domains by analyzing how users interact with their system. For each exchange between the system and the user, the log records the utterance waveform, the top N-best recognition hypotheses for that utterance, the semantic frame encoding the utterance, and the response presented to the user. A tool has been written at MIT that can be used for viewing this information and transcribing the utterances for training, using a web interface [29]. SpeechBuilder contains a slightly modified version of this tool, as shown in Figure 3-9.

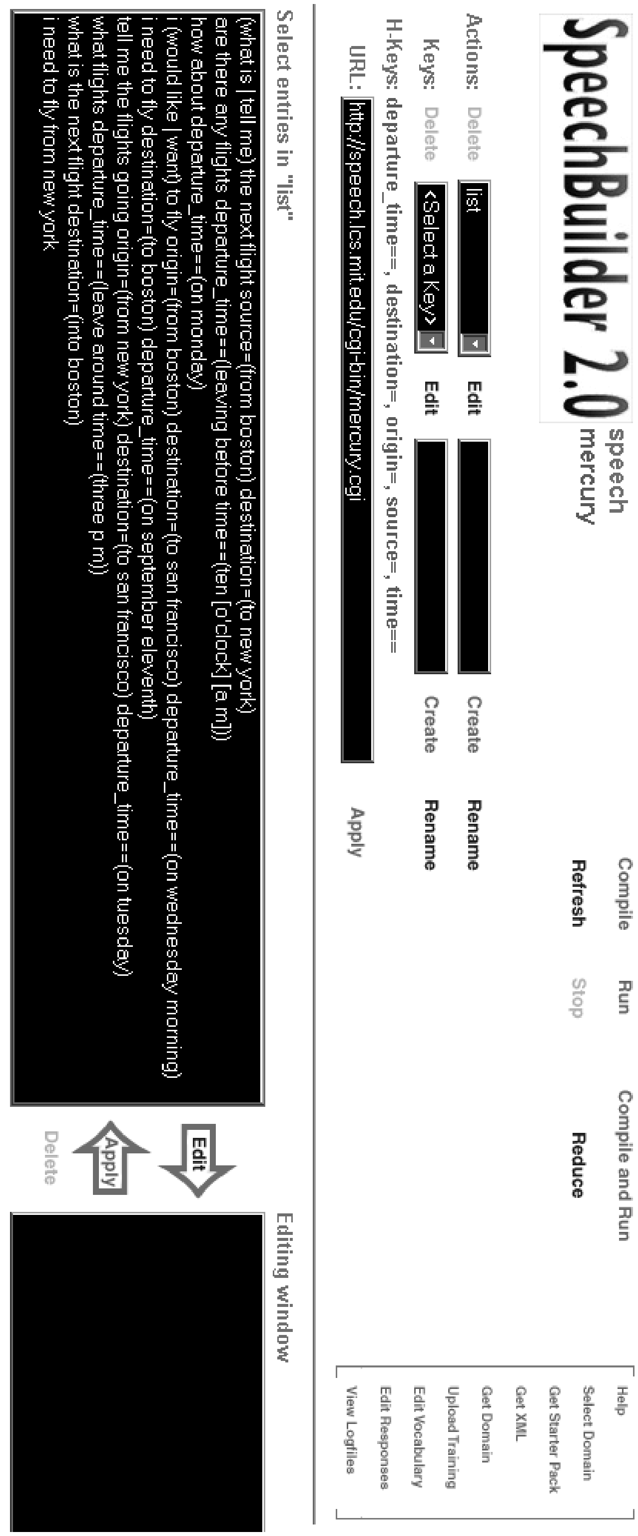

Figure 3-7: Web interface for SpeechBuilder. The screen shows editing example sentences for the "list" action.

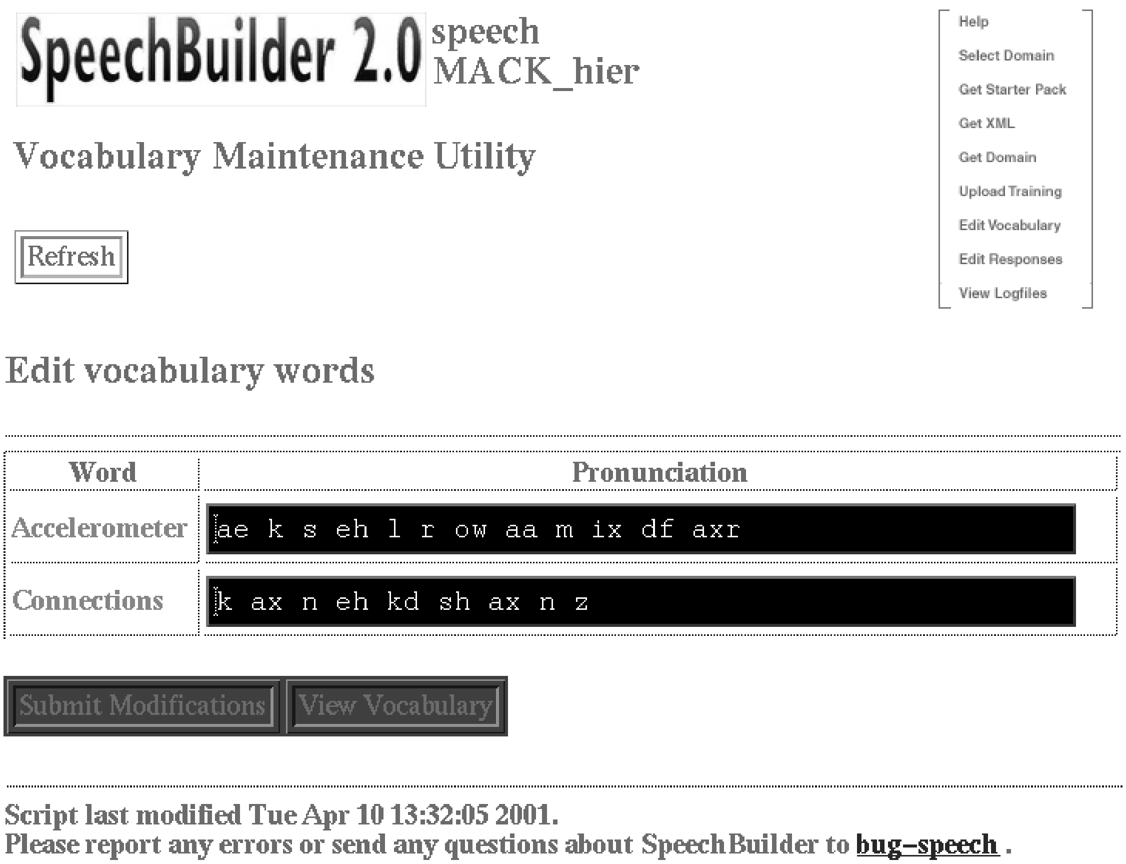

Figure 3-8: The SpeechBuilder vocabulary editing tool. The screen shows the developer editing the pronunciations of two words.

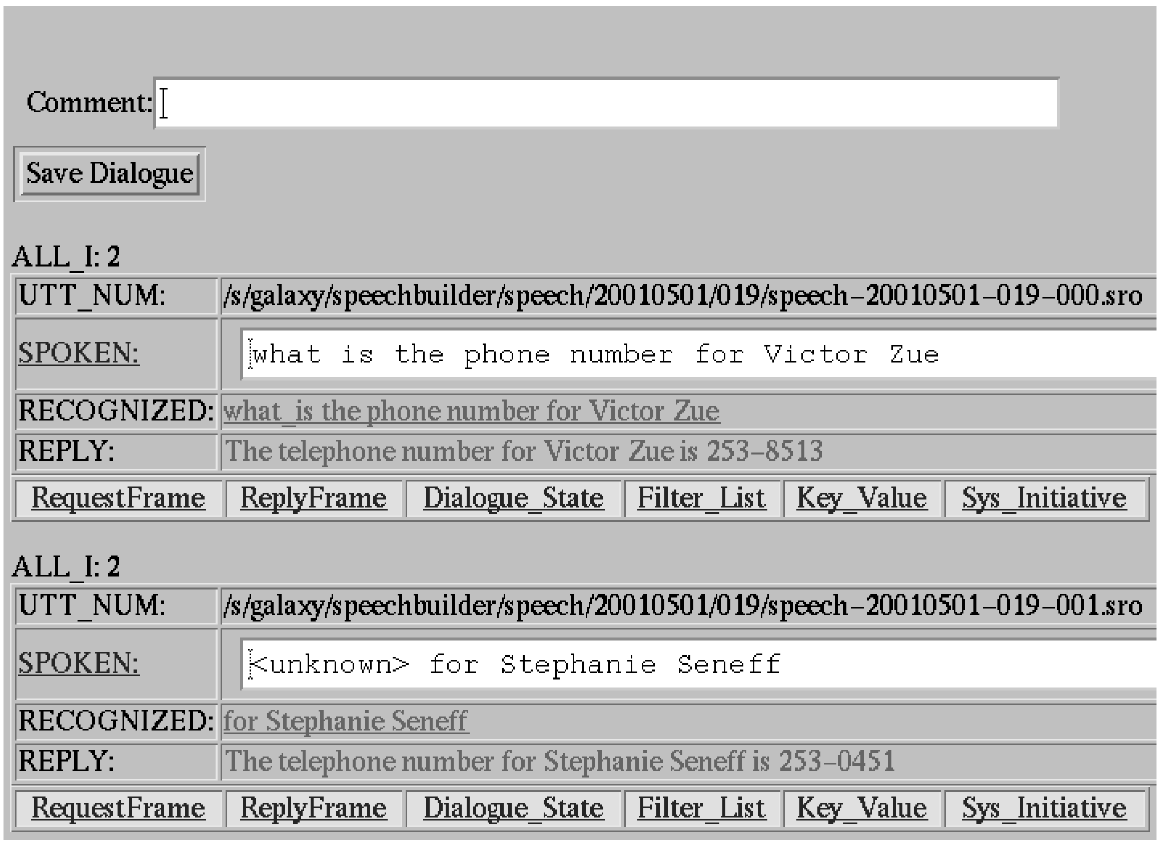

Figure 3-9: Log viewing tool used by SpeechBuilder. The screen shows a developer examining two utterances spoken by the user.

Chapter 4

Knowledge Representation

A developer configuring a domain using SpeechBuilder needs to model the concepts in the domain so that the proper grammars and generation templates can be built. This chapter addresses the knowledge representation used in SpeechBuilder to allow the developer to communicate domain specifics to the system. The first section describes the concept keys and sentence-level actions, the linguistic units that define the recognition and understanding templates. The second section discusses the model for connecting a database schema to the SpeechBuilder linguistic primitives, using accessor keys and properties.

4.1 Linguistic Concept Representation

In order to abstract the complexity of natural language parsing grammars from the developer, SpeechBuilder uses an example-based method of specification. The developers usually know, from intuition or experience, what queries the user may make of their system, and can use this knowledge to give example sentences, or "actions," for their domain. In addition, they can formulate the concepts, or "keys," that appear in their domain, and the values they may take on.

Based on this intuition, SpeechBuilder organizes knowledge into **concept keys**, and **sentence-level actions**. At times, these are simply referred to as "keys" and "actions." This part of the representation is inherited entirely from SLS-Lite [22].

Key	Examples
color	red, green blue
day	Monday, Tuesday, Wednesday
room	living room, dining room, kitchen

Table 4.1: Examples of concept keys

4.1.1 Concept Keys

Concept keys usually define classes of semantically equivalent words or word sequences. All the entries of a key class should play the same role in an utterance. Concept keys can be extracted from a database table (in the info model), or can be specified manually by the developer using the web interface. Table 4.1 gives some examples of concept keys.

Concept keys are used as non-terminal nodes in the natural language parsing grammar. The semantic frame is built using the concept keys that appear in a given parse of a sentence. Thus, in order to appear in a semantic frame a word *must* be a member of a concept key class.

SpeechBuilder allows concept keys to contain simple variations with the help of a few diacritics, which are adopted from JSGF [14]. Parentheses can be used to specify several alternatives, delimited by the "pipe," |, which all become valid values for this concept key (e.g. `(Robert | Bob) Jones`). Square brackets can be used to define parts of the concept key that are optional and can also contain pipe-delimited alternatives (e.g. `Boston [Massachusetts | Mass]`). Because some of the options defined using these diacritics can be very similar to one another, SpeechBuilder allows for regularization of the output value of these keys with the use of curly brackets, {}. So, for example, if the developer wants to represent all references to the city of Boston as `BOS`, they can form the concept key entry as `Boston [Massachusetts | Mass] {BOS}`.

4.1.2 Sentence-level Actions

Actions define classes of functionally equivalent sentences, so that all the entries of an action class perform the same operation in the application. Actions are used as top-level nodes in the natural language grammar, and thus the particular action name of a parsed utterance appears as the clause name in a semantic frame. A sentence can only parse into a single action class, so the sentences within different actions should not have any overlap. Table 4.2 gives some example actions.

A developer specifies actions by entering example sentences of that action, and SpeechBuilder generalizes on the example to build the natural language grammar. Example sentences within actions typically contain values of concept keys. SpeechBuilder generalizes all such sentences to accept all the entries within the particular key's class. For example, the last sentence in Table 4.2, "`Turn on the TV in the living room`," would be generalized into a template sentence as in Figure 4-1. At run time, this sentence, if spoken by the user, would result in the semantic frame given in Figure 4-2.

```
turn <onoff> the <appliance> in the <room>
```

Figure 4-1: Generalized sentence template from the example sentence "`Turn on the TV in the living room`"

```
{c set
   :onoff "on"
   :appliance "TV"
   :room "living room"}
```

Figure 4-2: Semantic frame for query "`Turn on the TV in the living room`" using generalized template shown in Figure 4-1

4.1.3 Concept Hierarchy

SpeechBuilder allows the slightly more advanced developer to build a structured

Action	Examples
identify	what is the forecast for Boston what will the temperature be on Tuesday I would like to know today's weather in Denver
set	turn the radio on in the kitchen please can you please turn off the dining room lights turn on the TV in the living room

Table 4.2: Examples of sentence-level actions

grammar when this is desired. This is done by using parentheses ("bracketing") to enforce a structure, possibly containing hierarchy, in the example sentences that appear inside the actions. The natural language parsing grammar is then generated according to this structure.

Bracketing is accomplished by enclosing a part of the sentence in parentheses and preceding it with a name and one or two equal signs (e.g. "`hier_key_name==(word1 word2 word3)`"). The name given to the bracketed portion of the sentenced is referred to as a **hierarchical key**, since it is treated similarly to a concept key in the NL parsing grammar.

Bracketed parts of sentences can contain either *strict* or *flattened* hierarchy. Strictly hierarchical bracketings preserve the complete substructure of the underlying concepts, and are indicated by the "`==`" symbol. Flattened hierarchical bracketings collapse any concepts contained within the hierarchical into the key class that is identified by the name of the bracketed expression. Table 4.3 contains examples of bracketed sentences and their resulting meaning representations (as encoded by the CGI parameter generation component, see the *Text Generation* part of Section 3.1.1).

4.2 Data Concept Representation

The info model for SpeechBuilder domains is designed to create a spoken language interface to a collection of structured data. The model calls for the data to be formulated as a table or tables in a relational database. At the current time,

Put object==(the blue box) location==(on the table) **object=(color=blue,item=box),location=(item=table)**
Put object=(the blue box) location=(on the table) **object=(blue_box),location=(table)**
Put the box location==(on the table location==(in the kitchen)) **item=box,location=(relative=on,item=table,** **location=(relative=in,room=kitchen))**

Table 4.3: Examples of strict (==) and flattened (=) hierarchy.

SpeechBuilder is only able to handle single-table domains, due to the complexity of the SQL generation grammars that are necessary to do SQL "JOIN" operations in multiple-table schema.

Since developers may not have access to a relational database SpeechBuilder allows them to upload the data using a comma-separated value (CSV) representation. This format can be generated by any spreadsheet program or by hand. Table 4.4 illustrates a sample data set for a domain to present information about people working in an organization, and Figure 4-3 gives the same data in CSV format.

Name	Phone	Email	Office
John Doe	123-4567	doe@foo.org	103
Bill Smith	123-9876	smith@bar.edu	107
Mary Jones	123-2222	mary@baz.com	257

Table 4.4: Data from the "info" domain

```
name, phone, email, office
key, people_property, people_property, people_property
John Doe, 123-4567, doe@foo.org, 103
Bill Smith, 123-9876, smith@bar.edu, 107
Mary Jones, 123-2222, mary@baz.com, 257
```

Figure 4-3: Data for the "Info" domain as a CSV file.

The data in Table 4.4 is used as an example throughout this section. In this application, each entry in the database has four columns. A user query may ask about

any of them (e.g., "`What is the phone number for Bill Smith`"). However, in most cases there will be some columns with information that won't be the subject of a query, called **properties** (e.g. it is not very common to ask "`what is the name of the person with the phone number 123-9876`" – so here *Phone* is such a property). We call the columns which are likely to be used to identify a row in the database **accessor keys**. So, since the database cells in the accessor key columns will be used in queries, *both the column values and the column names* must be incorporated into the speech recognition and natural language components. In contrast, for the property columns, *only the column names* need to be incorporated into these components.

Besides providing a name for each column the developer must specify whether the column is an accessor key or a property. Since there are often groups of properties that are essentially semantically equivalent from a natural language understanding viewpoint (e.g. *Phone* and *Email* – both tend to be asked about in the same way), SpeechBuilder allows the developer to define property grouping classes within the CSV file. For example, in this case *Phone*, *Email*, and *Office* are all grouped into the *people_property* group. The developer can then create one action class to access all of the keys in this group.

This method of specifying the information that a domain should access fits in with the Key-Action model described in section 4.1. Keys from the database table become keys in the SpeechBuilder domain, while the cell values from the table become the space for allowable values of their respective key. Each property group receives its own key in the domain, but only the column names appear as the key values, since the actual cell values are never vocalized by the user. For example, for the *people_property* group, there would be a *people_property* key in the domain with values "`phone`," "`email`," and "`office`". Based on information about accessor keys and properties in the CSV file, SpeechBuilder generates very crude example actions (see Section 5.3.1) to access the data (e.g., "`What is the phone for Bill Smith`"). The developer is then able to modify these actions to allow for more robust query handling for the domain.

4.3 Response Generation

In addition to configuring ways of asking about information, the developer needs to specify how the information should be presented to the user. Just as SpeechBuilder prepares generic example sentences for asking about information in a database table, it also creates generic responses for presenting this data (e.g. "`The phone is :phone`"). Here, the values received from database columns are indicated with the column name preceded by the colon (`:`). However, these responses tend to be very contrived because they are generated based simply on column headings, and it is hard for SpeechBuilder to determine which column in a database row should be used in the response. The response editing tool (shown in Figure 4-4) gives developers the functionality to modify the output of the system by editing strings such as the example given in this paragraph.

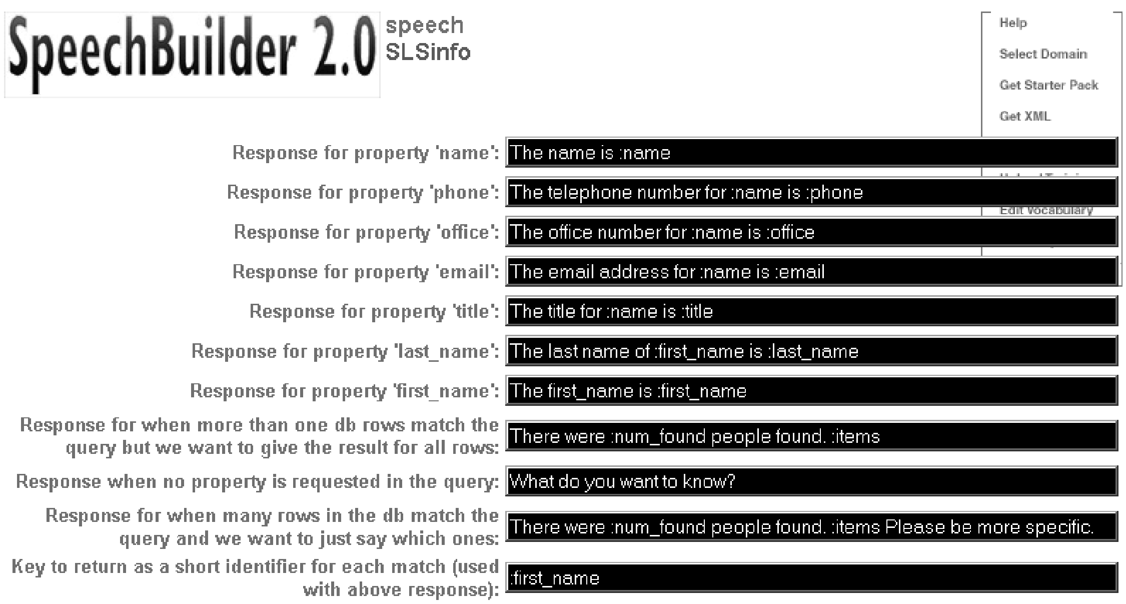

Figure 4-4: SpeechBuilder response editing tool

Because data tends to be presented differently across various columns, each database column has a set of responses corresponding to it. Namely, there are two responses per column – one to present the results of a successful query and another to inform the user that the query matched a database row, but the property asked for is not present in that row. In addition, there are a few generic replies which a developer

can configure, including a *welcome* and *goodbye* statement, as well as a statement when the system cannot understand what it is supposed to do.

The system needs to robustly handle cases when there are no matching rows in the database and when there is more than one row returned. SpeechBuilder provides fields for these responses, as well as some helper keys – *:num_found*, which contains the number of rows returned from the database, and *:items* which causes all items to be described when included. SpeechBuilder allows the developer to distinguish between when a "small" and a "large" amount of information is returned. When the amount of data is relatively small, it is appropriate to present the response for each of the rows, using the *:items* key (e.g. "`There are two people at LCS matching your query. The telephone number for Bob Jones is 111-2222 and the telephone number for Bob Smith is 222-1111`"). When there is a large amount of data, the domain will typically be configured to simply state the number of rows returned and ask the user to narrow their query (e.g. "`There are fifteen people who matched your query: Bob Jones, Mary Smith, ... Please be more specific.`").

Chapter 5

Development Issues

When a developer builds an application using SpeechBuilder, the internal process for configuring the various human language technology components is based on the techniques used by experienced developers to build such systems by hand. This chapter describes some of the features in SpeechBuilder that were more challenging to implement or automate, or that pose interesting generalization or machine learning problems. It focuses on issues in speech recognition and language understanding, response generation, and general infrastructure.

5.1 Language Model Overgeneralization

5.1.1 Problem Formulation

SpeechBuilder writes a tina natural language grammar for each domain. At runtime, this grammar is used to parse sentence hypotheses presented by the recognizer. In addition, tina is used to train a language model to constrain recognition hypotheses. SpeechBuilder uses a hierarchical n-gram [22, 18, 34] as a language model for speech recognition. Each concept key in an example sentence becomes a preterminal node in the n-gram [22]. Inside the keys, tina uses a word bigram to model the probabilities of accepting a certain value. In addition, tina uses a recursive transition network (RTN), which is similar to a bigram, to model the natural language gram-

mar constraints. This allows TINA to generalize on the training data to be able to cover previously unseen word sequences. However, in SPEECHBUILDER domains, this feature can also lead to overgeneralization in the natural language grammars. This sometimes leads to incorrect recognition hypotheses being accepted by the natural language server as valid parses.

Since the word bigram for each key is trained on the list of possible values for the key, the intuition is that only one of the values present in the training data will be accepted. However, due to the overgeneralization problem, occasionally this is not the case. For example, if a *city* key contains the entries "London England" and "New London Connecticut," the grammar learns that:

- London → England is a valid transition
- New → London is a valid transition and thus
- New → London → England is a valid transition combination

In this example, the grammar mistakenly learns that "New London England" is a valid value for the "city" concept key class, and therefore the language model is trained to accept this sequence. Thus, even though this value does not correspond to a valid city, a recognition error could lead to a sentence containing this value being accepted by the language model and the natural language grammar.

While this is a relatively minor and infrequently occurring error in most applications, it can actually be a very major one in a small set of domains. For example, in a domain that allows the user to spell proper names, TINA learns that most letter-to-letter transitions are acceptable. For example, in a sample data set of 1294 person first and last names in the LCSInfo domain, 487 out of a possible 728 transitions (67%) in the spelling bigram became valid alternatives. In actuality the average name in the data set contained 5.9 letters, or about seven transitions. So clearly, the overgeneralization problem is catastrophic in this domain.

5.1.2 Reducing Overgeneralization

SpeechBuilder addresses the overgeneralization problem by placing each value of a key in its own non-terminal node in the parsing grammar. Figure 5-1 illustrates a grammar with word spellings that suffers from the overgeneralization problem (for example, "Bob Jones j o n e s m o r e," with an invalid spelling, becomes a valid parse). Figure 5-2 is a modified version of the same grammar, which avoids this problem by introducing an extra level of non-terminals. Each of the values of the key *name* is now in a distinct non-terminal node. Since a bigram language model is trained for each node, the key values therefore no longer interfere with one another's bigram transition probabilities and natural language constraints.

An important note is that this solution undeniably results in a bigger, slower grammar (about four times slower on LCSInfo domain). However, usability experiments have showed that the overgeneralization problem presents a significant impediment to obtaining correct recognition hypotheses, and thus inevitably degrades the accuracy of the system. In SpeechBuilder, this is an especially significant problem because non-expert developers will not be able to understand what is causing this problem, nor will they be able to do anything to fix it. Thus, this is considered a worthwhile tradeoff for the purpose of this project.

```
.name
$Bob $Jones $j $o $n $e $s
$Mary $Smith $s $m $i $t $h
$John $More $m $o $r $e
```

Figure 5-1: Subset of a language parsing grammar that exhibits the overgeneralization problem (dollar signs indicate grammar terminal nodes).

Another solution was implemented but did not effectively solve the problem. This was to make separate non-terminal nodes for individual words based on their context in the sentence (in this case the word immediately to the left, the "left context," was used). This improved the situation marginally, but in the end overgeneralization still occurred because the long-range context of a word could not be modeled (e.g. "`Bob`

```
.name
name@@1
name@@2
name@@3

.name@@1
$Bob $Jones $j $o $n $e $s

.name@@2
$Mary $Smith $s $m $i $t $h

.name@@3
$John $More $m $o $r $e
```

Figure 5-2: The grammar of Figure 5-1 rewritten so as to avoid the overgeneralization problem (dollar signs indicate grammar terminal nodes).

`Jones j o n e s`" and "`Jonathan j o n a t h a n`" still yielded "`Jonathan j o n e s`" as a valid parse).

The above solution reduces the number of instances in which the overgeneralization problem is manifested. However, the problem will always be present in some form as long as bigram-type grammars are used. For example, even with the above solution, an example sentence like "`the lamp is on the table`" would allow the grammar to accept "`the lamp is on the lamp`." A more general solution is needed to more consistently control the overgeneralization problem. One way of doing this would be to use a context-free grammar such as JSGF [14] as the language model in the recognizer. This modification will soon be implemented in SpeechBuilder for comparison with the TINA-generated language model.

5.2 Hub Programs

The SpeechBuilder hub (see the *Hub* part of Section 3.1.1) executes one of several programs specifically written for SpeechBuilder. There are several modes of operation for SpeechBuilder domains that require more than one hub program to be

present for each domain. Since SpeechBuilder servers interact differently in the info model than in the control model, distinct hub programs exist for domains belonging to each of these two models. In addition, separate programs exist for running the domain on the MIT telephony hardware connected the developer access line and on the audio hardware of a developer local domain installation (see Section 5.6).

5.3 Creating Examples Based on Data

In the info model, the developer uploads a table of structured data to begin creating a domain. From this information, SpeechBuilder configures the linguistic concepts of the domain. This involves not only generating concept keys corresponding to database concepts (as described in Section 4.2), but also creating example sentence-level actions and responses for the domain. This section explains this process.

5.3.1 Sentence-level Actions

The data composes the base for the concepts present in the domain. However, it is not possible to automatically robustly figure out how a query about the data might be formulated, or how it should be presented to the user. SpeechBuilder attempts to formulate reasonable example queries and response templates based on column names and sample values extracted from the data. The developer can then edit the examples and put in additional action sentences to make the domain much more robust.

The data for the LCSInfo domain given in Figure 4-3 are used as an example throughout this section. SpeechBuilder needs to create example query sentences for each one of the columns in the database and for each one of the property grouping classes. The developer specifies which columns may be used to key a query to the domain (accessor keys), but it is hard to determine which of the accessor keys in a domain should be used to query for a particular column. Thus for the purpose of generating example query sentences, SpeechBuilder picks an accessor key at random. In the case of the LCSInfo domain, there is only one accessor key, *name*, so SpeechBuilder uses that one. The sentences are generated according to a typical

info domain query: "`What is the <property> for <key>`." In the LCSInfo domain, for the *email* column (*request_email* action), the example sentence is "`What is the email for John Doe`," which is actually a reasonable approximation for what the queries may look like.

In the case of property groupings (see Section 4.2), the example sentence generated for the action accessing the property grouping simply refers to one of the member properties. Because SpeechBuilder generalizes example templates and the property grouping name becomes a concept key in the domain, the action becomes valid for any of the member properties.

5.3.2 Responses

SpeechBuilder follows a similar process for generating sample responses. Since there is a response for presenting each one of the properties (see Section 4.3), selecting which property to return in the responses is not an issue. The significant problem is how to select which accessor key should be used to identify the particular row that matches the user query. One method of handling this is to pick an accessor key at random and use it to present the information. However, this can lead to misleading and just plain incorrect responses. For example, in the flight schedule domain, if *aircraft_type* is an accessor key, picking it would yield a response that looks like "`The 747 flight leaves at two p.m.`," which is not very informative. A much better response would include the *airline* and *flight_number* keys, e.g. "`American flight 252 leaves at two p.m.`" Due to this issue, SpeechBuilder does not include any accessor keys in the sample responses. Thus each response consists of a simple grammatical wrapper around a property, e.g. "`The departure time is two p.m.`" Of course, the developer is free to modify the responses, as is discussed in Section 4.3.

5.4 Pronunciation Generation

SpeechBuilder generates the recognition vocabulary from the keys and actions entered by the developer. SpeechBuilder obtains the pronunciations for domain

words from the 100,000-word LDC [16] PRONLEX dictionary that is used in most domains at MIT. However, at times the developer enters words that are not in the dictionary (often these are proper names). For these words, SpeechBuilder uses a text-to-phone conversion tool called `t2p` that is part of CMU's Festival speech synthesis package [4]. `t2p` is a good first-level approximation of pronunciations. It is able to achieve a 75% word correctness rate on the Oxford Advanced Learners Dictionary of Contemporary English (OALD), and 58% on a dictionary containing many more "foreign" words than the OALD [3]. The developer is able to adjust the pronunciation by using the vocabulary editing tool described in Section 3.2.2.

5.5 Registration

Each SpeechBuilder developer is required to register for a user account. In order to register to use SpeechBuilder, developers need to give some information about themselves using the registration tool (shown in Figure 5-3). The registration tool creates the domain directory structure for the developer and assigns a unique four-digit "developer ID" to be used by people calling in to talk to the developer's domains (see Section 5.6). In order to access the domain editing screens, the developer log into the system using browser authentication.

5.6 User Domain Access

SpeechBuilder gives developers two options for deploying their applications for actual use. The first option is to run the domain servers on MIT hardware and direct users to access the domain through the central domain access line. The second is to run the domain on a machine local to the developer, with no dependence on MIT hardware. This section describes these options.

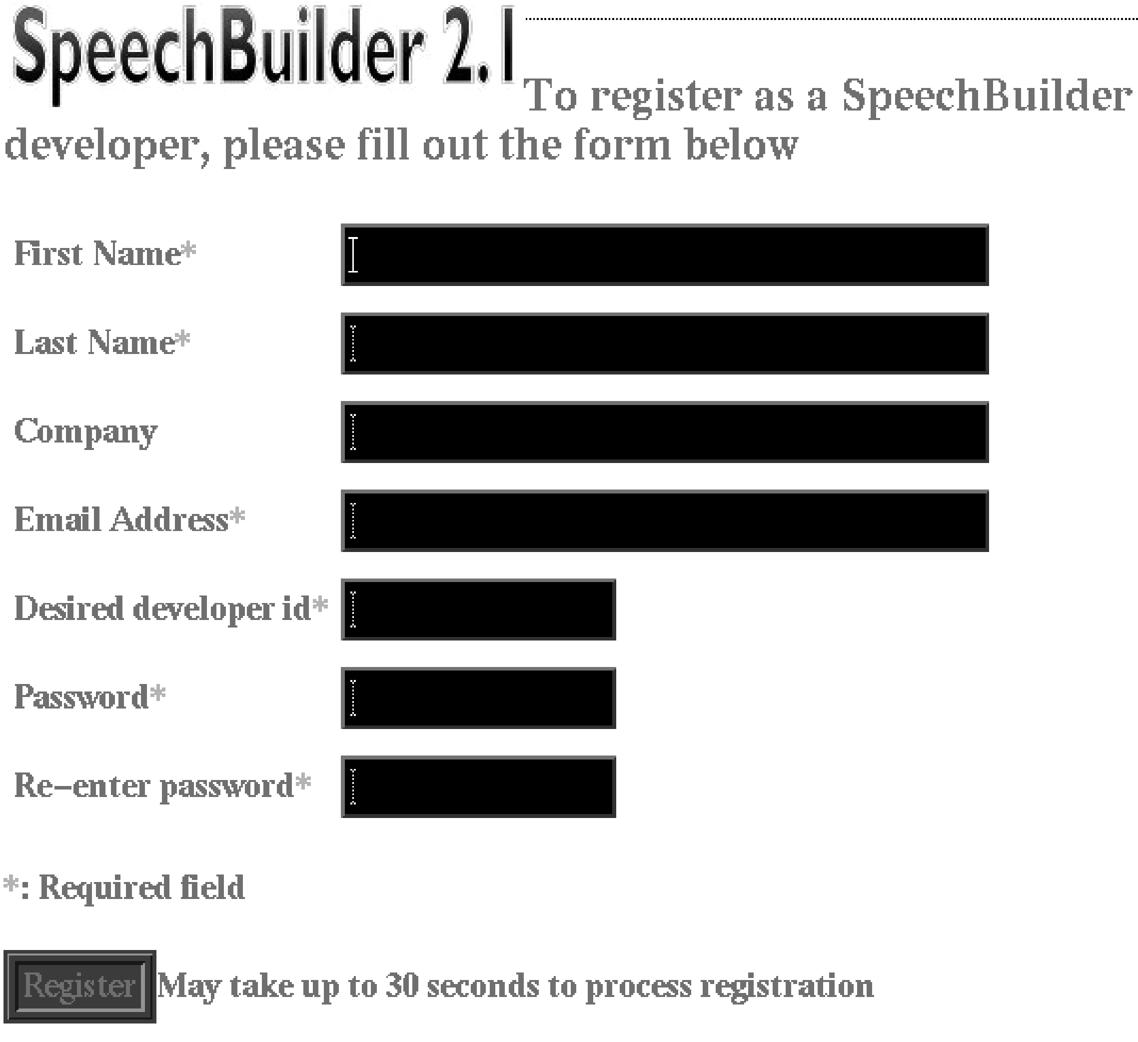

Figure 5-3: The SpeechBuilder developer registration tool.

5.6.1 Central Access Line

Most developers deploy their SpeechBuilder domains on telephony hardware located at MIT. The MIT installation of SpeechBuilder is connected to a developer access line – a telephone number which all developers call to talk to their domains. A developer starts the servers for a domain by clicking the "Start" button on the SpeechBuilder domain editing interface. When calling in, users say a four-digit identification number that was provided to the domain developer when he or she registered to use SpeechBuilder. In order to limit resources used by the runtime domain code, a developer is only allowed to run one domain at a time, so the caller

does not need to specify the domain they would like to talk to.

The developer access line is implemented using a new hub feature that allows hubs to transfer control of an online call between one another [7]. A central switch application receives the call, asks the caller for the developer ID, and using the hub transferring feature connects the call to the appropriate application. All developer domains are currently run on one machine connected to a single phone line. However, the hub switching code works across machines; so in the future it will be possible to scale the developer access hardware to several servers connected to many incoming lines.

Each developer's runtime domain software consists of all of the HLT servers and the developer hub. The servers of many developers need be able to coexist and be uniquely identifiable on the machine(s) running the server software. SpeechBuilder achieves this by assigning unique server port numbers to each developer. These port numbers are assigned at the time the developer registers with SpeechBuilder, and are stored in the scripts that are used to start the developer servers. This whole process is concealed from the developer.

5.6.2 Local Audio Deployment

In addition to the central access line, SpeechBuilder allows developers to download a complete set of server configuration files that allows them to run local installations of their domains. The "packaged" domain files that the developer receives are the same as the files used to run the domain on the central access line.

In order to be able to take advantage of this feature, there are several prerequisite steps that a developer must take. First, the galaxy software distribution must be installed on the machine that is to run the domain (this distribution is available to industry and government affiliates of the Spoken Language Systems Group of the MIT Laboratory for Computer Science). Second, the developer must download the generic acoustic models used by all SpeechBuilder recognizers, and install them on the machine that is to run the domain. Finally, the developer must set up a local audio configuration, such as a microphone and speakers connected to a sound card,

or a telephone and Skutch box (ring signal emulator) connected to a telephony card.

5.7 Echo Script

SpeechBuilder developers using the control model were often frustrated with having to write an entire back-end application in order to test even the most basic functionality of the HLT part of their domain. In addition, the developers wanted access to the full recognition hypothesis for each utterance in addition to the meaning representation of the query. A modification to the control model back-end server (see Section 3.1.3) allowed the full recognized string to be sent to the back-end script. Consequently, this allowed for the creation of a generic "echo script" that simply repeats the recognized utterance to the user. This script allows developers to check the performance of their domain's speech recognition and natural language parsing without writing a back-end script and finding a web server to host it. The echo script is now the default back-end script in control model domains.

5.8 Parse Tree Display

The TINA natural language parsing system is able to generate graphics of trees representing the nodes in the grammar that were used in the parse (such as the one shown in Figure 3-5). Jon Yi has implemented a CGI script that generates very similar displays based on a URL-encoded representation of the linear parse tree output produced by TINA. This tool is very useful in helping the developers visualize the parsing process that takes place in the natural language understanding components, and to debug domains containing hierarchy or complex parses. SpeechBuilder allows a developer to see a mapping of each example sentence to the "reduced" semantic frame representation [22]. Each sentence printed on the reduction screen is linked to the parse tree display tool with the specifics for that particular sentence included in the URL.

Chapter 6

Current Status

At the time of writing of this thesis, SpeechBuilder is deployed with all of the functionality working as described. SpeechBuilder began as version 1.0, which was basically the SLS-Lite system with some improvements to the natural language parsing grammars and the user interface, and developer registration and authentication. Version 1.1 brought more interface changes and introduced the developer access line and portable domains. Version 2.0 introduced the info model and all of the supporting tools (response editing, CSV file uploading, log file viewing, etc). Version 2.0 is the current "release" version; 2.1 is the "development" version which will include the ability to use languages other than English, and will allow more than one table in info model domains.

SpeechBuilder has been used in various environments (both inside MIT and out) to build a number of spoken language systems. The domains that have been built vary in complexity from somewhat contrived proof-of-concept applications, to fully functional and complex domains used for real applications. This chapter talks about some of this work.

The first section of this chapter discusses several fully-functional domains that have been used for demonstrations or have been deployed for actual users. The second section talks about some domains that have been built on a limited prototype level (some in as little time as one hour) to experiment with or demonstrate SpeechBuilder functionality.

6.1 Fully-functional Domains

6.1.1 LCSInfo and SLSInfo

LCSInfo is a phone-book type application for the more than 600 faculty, research staff, and students working at the MIT Laboratory for Computer Science and the Artificial Intelligence Laboratory. LCSInfo contains data about phone numbers, email addresses, group affiliations, room numbers, and positions for the people included in the domain. The data are updated automatically from the LCS personnel database that is used to generate the "directory" listing on the LCS web page, and manually from a flat file of AI Lab personnel. The domain can not only look up people's contact information, but can also connect the caller to a particular person's extension (this is similar to "auto-attendant" applications such as SpeechSite [32]). This was one of the first domains to be built using SpeechBuilder, so it implemented using the older control model. The back-end CGI script interfaces with the database and controls all dialogue and discourse phenomena.

A new domain called SLSInfo, having similar functionality to LCSInfo, has been built within the info model. SLSInfo only contains information about the people in the Spoken Language Systems Group. Even though this domain was built in the more constraining info model, its functionality is virtually identical to that of LCSInfo (except for the smaller data set, of course). This is encouraging because it shows that developers can create info model domains of similar complexity and flexibility as in the control model, without doing any programming.

In addition, an instance of the SLSInfo domain has been manually modified to use the envoice concatenative speech synthesizer that is being developed at MIT [36]. This is encouraging for eventually being able to give developers the option of using envoice as an optional synthesizer for SpeechBuilder domains (see Section 7.5).

6.1.2 SSCPhone

Auto-attendant domains similar to LCSInfo/SLSInfo have been built using SpeechBuilder outside of MIT. The most prominent example of such a domain is the phone book application at the Space and Naval Warfare Systems Center in San Diego, CA – called SSCPhone. The domain specifics in SSCPhone are very similar to those of LCSInfo. This domain allows users to access information about over 400 people working at the facility.

6.1.3 Office

The office domain is an application that lets people control the various hardware items occurring in a typical office using human speech. An actual installation of this domain has been implemented in the office of the Director of the MIT Laboratory for Computer Science [15]. The domain gives the user control over several devices (window blinds, projector, television, computer, VCR, projection screen, DVD, etc.) using simple speech commands. Since this domain involves external functionality, it uses the control model. The CGI script being used with this domain at LCS connects to the hooks controlling the various devices using a central switch made by Crestron Electronics [5].

6.1.4 MACK

The Media Lab Autonomous Conversational Kiosk (MACK) is a domain developed by the Gesture and Narrative Group at the MIT Media Lab as a speech-driven interface to a computerized character that answers visitors' queries about various Media Lab specifics [20]. MACK has been built by MIT Media Lab personnel with only minor help from experienced developers of speech-driven dialogue systems.

> The conversational kiosk is a guide that helps visitors to the Media Lab learn about what we do. A life-sized on-screen animated robot explains the lab's projects, groups, and consortia and gives directions about how to

> find them. The agent shares with the visitor a real physical model of the lab that the two participants can center their discussion around. Based on our previous research on multimodal dialogue systems and shared collaborative spaces, this system leverages people's natural language abilities to provide a richly interactive and easy-to-use kiosk. [17]

MACK allows visitors to ask about the various groups, projects, and consortia present at the Media Lab. It is also able to give people directions to various locations at the lab. The MACK domain contains over 20 distinct actions and information about over 40 groups, consortia and rooms.

MACK functions within the control model of SpeechBuilder, with a quite substantial back end application. The application controls the on-screen robot animation functions, handles discourse and dialogue management, queries a database of group information, and controls a separate speech synthesizer.

6.1.5 Weather

The weather domain is a "baby jupiter" domain. It was built with the goal of eventually having the same functionality as the successful jupiter weather information system [37]. The weather domain is even ahead of jupiter in the fact that it uses a real-time weather feed as opposed to a periodically updated data set. This domain contains information about over 500 cities in nearly 100 countries. It can report on any weather-related topic, such as precipitation, wind, humidity, cloud cover, etc.

Currently, the weather domain is implemented within the SpeechBuilder control model. This domain fits nicely into the newer info model, except for the fact that it requires the information lookup process to include a "JOIN" between several database tables. While SpeechBuilder does not yet support multiple tables, implementing this is a most immediate development goal. Once multiple table support is in place, a performance comparison between the weather domain and the hand-built jupiter will be possible. This comparison will be used to judge the robustness of SpeechBuilder domains.

6.2 Trial Domains

6.2.1 Schedule

The schedule domain is a dialogue system for accessing a class schedule for a meeting of the industry affiliate user group of the Spoken Language Systems Group at LCS. The domain is implemented within the info model. It contains information about a full-day schedule of events and their details. It has information about the instructor(s) of each activity, the class start and end times, and the class content. This domains utilizes the SpeechBuilder bracketing facility (see Section 4.1.3) to allow the user to make relatively complex queries such as "`Who teaches the class that ends at one?`" and "`When does the class taught by Jim start?`"

The Schedule domain is a proof-of-concept domain that was implemented by Chao Wang as part of a SpeechBuilder laboratory evaluation. With a relatively small amount of work, this domain could be adapted to serve as a daily personal schedule access application for any end user.

6.2.2 Stocks

The stocks domain is a demo system for accessing online stock quotes. This system contains information about the stock price, day highs and lows, and bid and ask prices, for a small group of stocks (less than ten). The stocks domain is implemented in the info model with a table of static data. This domain was built by a group of engineers from Fidelity Investments as part of the SpeechBuilder laboratory at an industry affiliates meeting in about an hour's time.

6.2.3 Flights

The flights domain was built largely to test the hierarchical bracketing feature of SpeechBuilder (see Section 4.1.3). It was designed to simulate the MERCURY flight reservation system [28], but in this implementation only accepts simple flight queries – with no actual reservation transaction functionality. This domain was implemented

in the control model, and was never tested with any real back-end script (one similar to the echo script described in Section 5.7 was used). This domain handles queries about flights using as little or as much information as the user is ready to provide.

Chapter 7

Conclusions

SpeechBuilder, which was implemented as part of this thesis project, extends on the SLS-Lite system [22] in many directions, improving the overall infrastructure and the use of the underlying human language technology. SpeechBuilder is a first attempt at a complete tool for development of robust mixed-initiative spoken language systems that does not require developer expertise or application programming. While almost every single component of the original SLS-Lite system has been modified somehow as part of this work, SpeechBuilder improves over SLS-Lite most significantly in the following areas:

1. **Knowledge Representation**. SpeechBuilder introduces the info model for application development, which allows a developer to configure an information-access application without doing any programming.

2. galaxy **Component Use**. SpeechBuilder uses the full set of human language technology components, similar to the spoken language systems previously built at MIT.

3. **Infrastructure**. SpeechBuilder contains a developer registration and authentication scheme to allow for access control. In addition, SpeechBuilder introduces the developer access line and gives developers the ability to run local installations of their applications at their home sites.

4. **Language Generation**. SpeechBuilder, in the info model, uses the language generation component extensively. For each domain, SpeechBuilder generates fairly complex language generation templates for several distinct uses.

5. **Speech Recognition and Natural Language Understanding**. SpeechBuilder introduces the out-of-vocabulary model for speech recognition. In addition, SpeechBuilder patches the TINA grammar overgeneralization problem and thus improves the accuracy of natural language parsing.

While SpeechBuilder improves on these areas, there is still much work to be done in this field. The rest of this section discusses the ongoing and planned future work on SpeechBuilder.

7.1 Dialogue Control

The current SpeechBuilder info model dialogue control manager uses a generic hard-wired protocol for handling domain dialogue. In the future, this component should increase in complexity and allow the developer more control over domain situations. One improvement that can be made is to allow for automatic disambiguation of incomplete queries (e.g., user: "`I want to fly to New York tomorrow`," system: "`Okay. What city will you be leaving from?`"). Another possible improvement is to allow the user to actually modify information in the database, so as to allow for more transaction-based domains (such as flight reservations) to be implemented in the info model. Yet another change is to give the developer control over the dialogue situations, i.e. specifying heuristics for when disambiguation is needed, when there is too much data to be spoken, etc.

7.2 Communication Protocol

The current CGI protocol for communication with back-end applications in the control model can be improved. A web server runs each invocation of a CGI script in a

separate process using a separate socket connection. In addition, a CGI script has no state from invocation to invocation. For these reasons, CGI is bulky and inefficient for real-world applications.

A new communication protocol based on remote procedure call (RPC) conventions is desirable. Work is currently being done to implement a Java API for connection to remote applications running continuously on a single process, with state information maintained in runtime memory space [8].

7.3 Confidence Scoring

SpeechBuilder recognizers currently do not have the ability to estimate the level of confidence of a recognition hypothesis (that is, how certain the recognizer is that it matches the corresponding acoustic evidence). Confidence scoring is a desirable feature, especially if control of various confidence scenarios can be given to the developer (e.g., do not accept a user utterance if the confidence score is too low). Adding confidence scoring would also allow for unsupervised training of acoustic and linguistic recognition models (see section 7.4).

7.4 Unsupervised Training

The acoustic models currently used in SpeechBuilder recognizers are based on generic telephone data collected in mainstream MIT domains. The language models are trained on the example sentences that the developer provides in the sentence-level actions, and there is no feedback loop to allow the data collected in the operation of SpeechBuilder domains to be used to improve these models. With the confidence scoring in place, a more long-term goal is to incorporate unsupervised training of acoustic and linguistic models based on data recorded during the runtime of the domain recognizer.

7.5 Synthesis

The current implementation of SpeechBuilder uses the DECtalk commercial speech synthesizer [9], which provides speech synthesis of reasonable, but not exceptionally high, quality. The MIT concatenative speech synthesizer, envoice [36], would be a significantly better-sounding solution. However, since an envoice synthesizer is based on a corpus of speech samples specific to the domain it is being used for, the developer will need to do some work before being able to use envoice in a given domain. It would work well to give the developer the option of using envoice if there is time to record a synthesis corpus (perhaps using the developer's own voice), with DECtalk being the default option.

7.6 Multi-lingual SpeechBuilder

SpeechBuilder by convention has so far only addressed domains dealing with speech in American English. However, domains in other languages have been implemented using the galaxy framework [35, 25], and thus it is desirable to allow SpeechBuilder developers to build domains in languages other than English (of course a basic requirement is that a speech recognizer for the language be available). There have been several efforts to "port" SpeechBuilder to non-English languages. In the future, these ports will be integrated into SpeechBuilder. The domain selection screen will allow the developers to select a language for a domain. While the domain editing interface will be in English regardless of the domain language, the recognition and synthesis components will be specific to the language selected for each domain. The rest of this section describes the efforts to date to port SpeechBuilder to Japanese and Mandarin Chinese.

7.6.1 Japanese – JSpeechBuilder

A Japanese port of SpeechBuilder has been implemented in cooperation with Dr. Mikio Nakano, a visiting scientist from NTT, and John Yi [19]. The difficulties of

making this port have been allowing entry of Unicode characters on SpeechBuilder domain editing screens, and phonological analysis of Japanese phonetic character input.

Most of the SpeechBuilder code was already compatible with Unicode characters. However, the XML parser used within SpeechBuilder (`XML::Simple` module) was not readily compatible and required a small modification. Japanese is a syllable-based language, so the example sentences entered by the developers do not typically have explicit word markings. However, a word-level language model in the recognizer performs much better than a syllable-level one. Thus, phonological analysis needs to be performed on the sentences to identify the words. JSpeechBuilder uses a phonological analyzer to segment the phoneme sequence for a sentence or key into probable words.

7.6.2 Mandarin Chinese – CSpeechBuilder

The Mandarin Chinese port of SpeechBuilder has been implemented in cooperation with a group of visiting scientists from Delta Electronics, Inc. under the leadership of Dr. Lynn Shen. CSpeechBuilder uses a phonetic Romanization of Chinese language utterances, and thus it does not need to use Unicode. CSpeechBuilder does not attempt to solve the phonological analysis problem. Rather the developer to identify words by joining sub-word units with underscores (e.g. "`dian4_deng1`" meaning "light").

CSpeechBuilder has been used to build several demonstration domains, including an application to control various appliances in a futuristic home at Delta Electronics [31].

7.7 Database Schema

The range of database schema that the info model can currently handle is very limited. All database cell entries are treated as strings and there is no support for multiple tables. In the future, SpeechBuilder should be able to support numeral and boolean

data types in the database. The numeral data type will allow for test queries such as "`Show me all the flights after five.`" The boolean data type will enable SpeechBuilder domains to handle yes/no questions such as "`Does that flight serve dinner?`"

Multiple table support is another essential improvement, as it will enable SpeechBuilder to handle domains where the data is spread over several database tables. For example, in jupiter there is a "geography" table containing city-to-airport-code mappings, and a "weather" table containing forecasts keyed on airport codes [37]. Once this change is made, SpeechBuilder will be able to build a weather domain using the same database schema as jupiter. This is an important goal for purposes of evaluating SpeechBuilder domain performance – since this will allow for comparison the SpeechBuilder weather domain to jupiter based on the standard evaluation data sets [37].

Bibliography

[1] L. Baptist and S. Seneff. GENESIS-II: A Versatile System for Language Generation in Conversational System Applications. In *Proc. ICSLP*, Beijing, 2000.

[2] I. Bazzi and J. Glass. Modeling Out-of-vocabulary Words for Robust Speech Recognition. In *Proc. ICSLP*, Beijing, 2000.

[3] A. Black, K. Lenzo, and V. Pagel. Issues in Building General Letter to Sound Rules. In *Proc. ESCA Speech Synthesis Workshop*, Jenolan Caves, 1998.

[4] A. Black and P. Taylor. The Festival Speech Synthesis System. Technical Report HCRC/TR-83, Human Communication Research Centre, 1997.

[5] Crestron Electronics, Inc. http://www.crestron.com/.

[6] CSLU Toolkit. http://cslu.cse.ogi.edu/toolkit/.

[7] S. Cyphers. Personal communication, 2001.

[8] S. Cyphers and J. Smoler. Personal communication, 2001.

[9] DECtalk. http://www.forcecomputers.com/product/dectalk/dtalk.htm.

[10] E. Filisko. Personal communication, 2001.

[11] J. Glass, T. Hazen, and L. Hetherington. Real-Time Telephone-Based Speech Recognition in the JUPITER Domain. In *Proc. ICSLP*, Beijing, 2000.

[12] J. Glass and E. Weinstein. SPEECHBUILDER: Facilitating Spoken Dialogue System Development. Submitted to Eurospeech, Aalborg, 2001.

[13] T. Hazen. Development of the VOYAGER system. *MIT Laboratory for Computer Science Spoken Language Systems: Summary of Research*, pages 20–22, 1999.

[14] Java Speech Grammar Format Specification. http://java.sun.com/products/java-media/speech/forDevelopers/JSGF/.

[15] M. Ladd and N. Meyerhans. Personal communication, 2001.

[16] Linguistic Data Consortium. http://www.ldc.upenn.edu/.

[17] MACK: Media Lab Autonomous Conversational Kiosk. http://gn.www.media.mit.edu/groups/gn/projects/kiosk/index.html.

[18] M. McCandless and J. Glass. Empirical Acquisition of Language Models for Speech Recognition. In *Proc. ICSLP*, Yokohama, 1994.

[19] M. Nakano and J. Yi. Personal communication, 2001.

[20] Y. Nakano and J. Huang. Personal communication, 2001.

[21] Nuance Communications, Inc. http://www.nuance.com/.

[22] J. Pearlman. SLS-LITE: Enabling Spoken Language Systems Design for Non-Experts. Master's thesis, MIT, 2000.

[23] Philips GmbH, Aachen. *SpeechMania 7.1 Developers Guide*, 2001.

[24] S. Seneff. TINA: A Natural Language System for Spoken Language Applications. *Computational Linguistics*, 18(1), 1992.

[25] S. Seneff, J. Glass, T.J. Hazen, Y. Minami, J. Polifroni, and V. Zue. MOKUSEI: A Japanese Spoken Dialogue System in the Weather Domain. *NTT R&D*, 49(7), 2000.

[26] S. Seneff, E. Hurley, R. Lau, C. Pao, P. Schmid, and V. Zue. GALAXY-II: A Reference Architecture for Conversational System Development. In *Proc. ICSLP*, Sydney, 1998.

[27] S. Seneff, R. Lau, and J. Polifroni. Organization, Communication, and Control in the GALAXY-II Conversational System. In *Proc. Eurospeech*, Budapest, 1999.

[28] S. Seneff and J. Polifroni. Dialogue Management in the MERCURY Flight Reservation System. In *Proc. ANLP-NAACL Satellite Workshop*, Seattle, 2000.

[29] S. Seneff and J. Polifroni. Hypothesis Selection and Resolution in the MERCURY Flight Reservation System. In *Proc. Human Language Technology Conference*, San Diego, 2001.

[30] S. Seneff, J. Polifroni, and P. Schmid. PEGASUS: Flight departure/arrival/gate information system. *MIT Laboratory for Computer Science Spoken Language Systems: Summary of Research*, pages 25–26, 1998.

[31] L. Shen. Personal communication, 2001.

[32] SpeechWorks International, Inc. http://www.speechworks.com/.

[33] Voice eXtensible Markup Language version 1.0. http://www.w3.org/TR/voicexml/.

[34] C. Wang. Personal communication, 2001.

[35] C. Wang, S. Cyphers, X. Mou, J. Polifroni, S. Seneff, J. Yi, and V. Zue. MuXing: A Telephone-Access Mandarin Conversational System. In *Proc. ICSLP*, Beijing, 2000.

[36] J. Yi, J. Glass, and L. Hetherington. A flexible, scalable finite-state transducer architecture for corpus-based concatenative speech synthesis. In *Proc. ICSLP*, Beijing, 2000.

[37] V. Zue, S. Seneff, J. Glass, J. Polifroni, C. Pau, T. J. Hazen, and L. Hetherington. JUPITER: A Telephone-Based Conversational Interface for Weather Information. *Proc. IEEE Trans. SAP*, 8(1), 2000.